Research report 188

SYNTAX ERROR HANDLING IN LANGUAGE TRANSLATION SYSTEMS

JULIA DAIN

(RR188)

Compilers, interpreters, editors are examples of language translation systems which are an essential component of computing. Such systems need to be able to handle most kinds of input, including input that is syntactically incorrect, according to the context-free grammar rules for the system. This paper surveys syntax error handling in theory and practice. Two theoretical models for syntax errors are minimum distance errors, based on the number of operations needed to transform an incorrect string into a correct one, and parser defined errors, based on correct prefixes. Constraints on languages that can be handled are given and different parsing methods are compared. Error reporting is important because it is the user interface for error handling. Good error messages are precise, informative and constructive. Concepts and techniques for error recovery schemes are explained. Goals for good error recovery are set and practical, implemented schemes are evaluated and their performance compared. A minimum standard of acceptable performance on three-quarters of a data set of Pascal programs is set. Future directions for research are indicated, with particular reference to the problems posed by error handling for interactive programming environments.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

June 1991

CONTENTS

INTRODUCTION

Language translation systems such as compilers and interpreters for programming languages have the task of translating input in a source language into equivalent output or actions in a target language. In addition to performing translation of correct input, language translation systems are also usually expected to handle input that does not conform with the legal rules or definition of the source language. Such incorrect input is said to contain lexical, syntactic or semantic errors, according to whether the lexical, syntactic or semantic rules of the source language are violated.

A language translation system is conventionally decomposed into several tasks or phases including lexical analysis, syntax analysis, semantic analysis, code generation and code optimisation. Lexical analysis apportions the input stream into those strings of input symbols that make up the lexical tokens of the source language. Syntax analysis checks that the sequence of lexical tokens accords with the syntactic structure of the source language, usually described by a context-free grammar (CFG). Semantic analysis checks for non-context-free structure. Code generation and optimisation perform the desired translation into target code. The phases of lexical, syntactic and semantic analysis are usually expected to handle lexical, syntactic and semantic errors respectively. Thus a parser performing syntax analysis is usually expected to handle any string of lexical tokens, including strings that cannot be parsed according to the CFG used to describe the source language. The design of a syntax analyser includes design decisions on how to proceed when the input is syntactically incorrect: the resulting procedures form a syntax error handling scheme.

Good syntax error handling enhances the value of a translation system by showing its users how to provide correct input; bad error handling prevents effective use of a system. Syntax analysis is well understood for correct input, to the extent that designers frequently use software tools for the automatic construction of efficient syntax analysers. But the analysis of syntactically incorrect input remains less well understood. This survey is concerned with the design and construction of good syntax error handling schemes for language translation systems. Why it is important to have good error handling can be illustrated with an example. Consider the following source input program in the C programming language:

```
1    main()
2    {   int i;
3        i = 1
4        printf("%d\n", i);
5    }
```

There is a semi-colon missing from the end of line 3. When this program is submitted to the portable C compiler widely available under the UNIX system [Johnson 1978], the diagnostics issued are:

```
"file.c", line 4: syntax error
"file.c", line 4: illegal character: 134 (octal)
"file.c", line 4: cannot recover from earlier errors: goodbye!
```

These messages are misleading, incomprehensible, and unhelpful. The error is not located

precisely - the user's attention is directed to the whole of line 4. The message about an illegal character cannot be deciphered without a character code table. The word "illegal" should not be used, as it is inaccurate and intimidatory [Molich and Nielsen 1990]. No hints are given as to what might constitute a correction. Finally the compiler appears to give up altogether. The error handling mechanisms are inadequate and prevent the compiler from doing a good job on the input, which contains a common, simple error. Giving the same input to the GNU C compiler [Stallman 1989] results in the error messages:

```
"file.c": In function main
"file.c":4: parse error before `printf'
```

These messages are an improvement over the previous example, as they locate the error precisely, and do not give any misleading or inaccurate information. They would be further improved by giving the user some suggestions on how to correct the error.

The issues in syntax error handling can be summarized by the following questions: what is a syntax error? how will it be dealt with? which languages will be handled? and what are the technical issues for the parser? The first question, what is an error, can be answered formally, by giving a theoretical model which characterizes errors abstractly, or informally, from a practical or user-oriented view which characterizes errors according to plausibility or frequency of occurrence. The second question, how will an error be dealt with, can be answered either by halting syntax analysis as soon as an error is detected, or by attempting to continue analysis by some means of recovery. Recovery must restart the parser and may include alteration or repair to the input. It may not be feasible to produce a theoretically optimal repair, because of considerations of efficiency, nor may it be feasible to determine the original intention of the user, because of the nature of the system in which the recovery method operates. The third question is whether the languages to be handled are arbitrary context-free languages or a restricted class, such as practical programming languages. Typical programming languages may exhibit special characteristics which have implications for error handling. Studying such languages, and the use of them, may lead in different directions from a theoretical study: for example, to questions such as what are the psychologically plausible errors or the most common in practice, which errors are hard to handle, and what are the implications for programming language definition. The fourth question addresses the technical issues for a parser: how will errors be identified, how will recovery be effected, what communication with the user will take place. Relevant factors include the method used for parsing, the nature of access to the source input, the extent to which strategies are language-dependent, and the need for efficiency.

This paper surveys theory and practice of syntax error handling, addressing the particular issues of theoretical models for syntax errors, languages to be handled, error detection, error reporting, and practical methods for error recovery. Some familiarity is assumed with the principles and practice of compiler design.

The terms error handling, error recovery, error repair and error correction are used with different meanings in the literature. In this paper the following definitions will be used. *Error handling*, or an *error handling scheme*, is a general term for the method used to treat a syntax

error in the input to a parser, including reporting the error to the user. *Error recovery* is error handling in which an attempt is made to continue parsing the remaining input after detection of a syntax error. *Error repair* describes those changes which an error recovery scheme may make to the input and possibly the parse stack in order to continue parsing. *Error correction* is error handling which alters the input to that which the user intended.

1. MODELS FOR SYNTAX ERRORS

Before discussing how to handle syntax errors, it is necessary to establish with some precision what a syntax error is. Unfortunately, an informal concept of a syntax error does not always correspond with a formal definition. A CFG used to describe the context-free syntax of a programming language defines precisely those strings which are syntactically correct programs, namely sentences of the language, but not what the syntax errors are for a string which is not a sentence of the language. This section reviews the models of minimum distance errors and parser defined errors.

1.1 Minimum Distance Errors

Minimum or Hamming distance is frequently used as a formal model for syntax errors. It measures the shortest way to transform a syntactically incorrect program into a correct one and thus approximates to the programmer's concept of syntax errors. Given a fixed set of edit operations, typically the insertion, deletion or replacement of a single symbol, the minimum distance between two strings is the minimum number of edit operations needed to transform one string into the other. For example, the minimum distance between the two strings of parentheses "(())" and "()" is two, obtained by two insertions. Additionally, costs may be associated with the edit operations to give a least-cost sequence of edit operations needed to transform one string into the other [Wagner and Fischer 1974]. The minimum distance measure is used to define a minimum distance error correction for a string and a language: a minimum distance error correction is a sentence of the language nearest to the string, in the sense that there is no other sentence whose minimum distance from the string is smaller. A least-cost error correction is similarly defined to be a sentence of the language such that there is no other sentence whose least-cost sequence of edit operations is of smaller cost. For syntax error correction, the symbols to be operated upon by the edit operations are the terminals of a CFG (the lexical tokens of a programming language).

The minimum distance error correction for a string and a language is not uniquely defined. Consider the string "(()" and a language of balanced parentheses generated by the grammar with productions $S \rightarrow () \mid (S)$. Two minimum distance error corrections of distance 1 for this string are "()" and "(())". This example is readily extended to that of common programming languages. For example, in Pascal [Cooper 1983], arithmetic expressions may be bracketed with an arbitrary number of pairs of parentheses ( ), and in C [Kernighan and Ritchie 1978], blocks may be bracketed with an arbitrary number of pairs of braces { }.

A minimum distance error correction for a string locates the errors in a string at the points at

which the edit operations are applied. The location of the leftmost error in an incorrect string can be uniquely defined for a given minimum distance error correction as the leftmost point at which the string differs from its correction. For the example string "(()" and the correction "()", the leftmost error is at the second symbol. For a string with more than one correction we may wish to choose a correction with the longest possible prefix of the original string. So for the string "(()" and the corrections "()" and "(())", the correction with the longest possible prefix is "(())". The leftmost error location for a longest prefix minimum distance error correction then uniquely defines a point in an incorrect string which is the *first minimum distance error*. For a string and a longest prefix minimum distance error correction at minimum distance one, there is only one point at which the string differs from the correction and this is the (unique) *minimum distance error*. For the string "(()" the minimum distance error is after the third symbol, at the end of the string.

1.2 Parser Defined Errors

A parser for a language may not detect an error in its input until some distance after the location of the leftmost error according to the minimum distance error model. The following is an incorrect Pascal program in which the plausible syntax error is the omission of the keyword 'for' at the beginning of line 3. (In the Pascal programs used as examples, declarations of variables are omitted; these omissions constitute semantic errors, not syntax errors.)

```
1    program example(output);
2    begin
3        n := 1 to 10 do sum := sum + n
4    end.
```

The minimum distance error coincides with the plausible error and occurs on line 3 after the symbol 'begin' at the symbol 'n'; a minimum distance error correction for the program is to insert the symbol 'for' at this point. But parsers such as the practical LL or LR parsers with one symbol of lookahead will not detect an error until the symbol 'to' is met, because the fragment 'n := 1' forms a legal Pascal statement. Peterson introduced the concept of parser defined errors to model this class of errors [Peterson 1972]. The parser defined error in an incorrect string, with respect to a language, marks the point at which a prefix of the string ceases to be a prefix of the language. In the example above, the string

```
program example(output); begin  n := 1
```

is a prefix of a Pascal program, but the string

```
program example(output); begin  n := 1 to
```

is not. The parser defined error is at the symbol 'to'. The name "parser defined error" is somewhat misleading, as the parser defined error is defined by the language and its prefixes, rather than by a parser for the language.

It must be borne in mind that error handling is performed at the level of lexical tokens, the units or symbols for syntax analysis, and not at the level of individual source characters. The human who reads the source program makes use of much more information than is available to a

parser or syntax error handling scheme, including not only composition of tokens from source characters, but also layout of the program, context-sensitive information such as types, and logical meaning. So the model of minimum distance error correction, although theoretically useful and convenient as a formal measure of the number of errors in a string, is limited in its ability to model the user's intention. The model of parser defined errors is useful in the way that it models the errors in a string from the point of view of a parsing machine, but is even more limited in its ability to model the human user's intention.

2. LANGUAGES FOR ERROR HANDLING

This sections surveys work on programming language design related to the occurrence and handling of syntax errors, not the much wider questions of language design either in general or related to other aspects such as programming languages for particular applications. There are few papers on this topic in comparison with work on the design of error handling schemes. There is some work on the design of languages with respect to the kind and frequency of occurrence of errors [Peterson 1972; Gannon and Horning 1975; Wetherell 1977; Ripley and Druseikis 1978]. A few papers are concerned with various measures of size of CFGs [Ginsburg and Lynch 1976; Bertsch 1983]. Finally several authors consider ways of making CFGs more suitable for use in particular error handling schemes [Pai and Kieburtz 1979, 1980; Tai 1980a, 1980b; Mauney and Fischer 1988].

Gannon and Horning [1975] discuss the design of programming languages from the point of view of improving reliability of programs. Both syntactic and semantic errors are considered and it is shown that language design decisions have a significant effect on progam reliability. This work does not seem to have been followed up; current research in software engineering for reliability favours the use of formal methods and development of programs from specifications, via transformations.

Some properties which are desirable for error correction are undecidable for arbitrary CFGs; in particular, it is undecidable whether minimum distance errors and parser defined errors coincide [Peterson 1972; Wetherell 1977]. However, strings for common programming languages can be exhibited in which the minimum distance error does not coincide with the parser defined error. (For Pascal, see the example program given in in Section 1.2.) A probabilistic approach can be used to give probabilistic measures for the occurrence of errors and the distance from an error to its detection by a parser [Wetherell 1977]. Measures can also be used for the probability that an error will be detected and that a repair is a prefix of the language. Wetherell suggests that language designers should use such probabilistic analyses in an attempt to make languages less susceptible to common errors. Ripley and Druseikis [1978] give an analysis of a collection of students' Pascal programs which shows the nature of such common errors for Pascal. They find that the majority (88%) of syntax errors are simple, the most common being omission of a single token and substitution of an incorrect token, and that errors occur infrequently. They discuss the relationship between such errors and language constructs. Spenke et al [1984] remark that misuse of keywords, comment delimiters, and string quotes

causes great problems to all methods for error recovery. They recommend avoidance of such problems by appropriate language design, for example by restricting comments and string constants to a single line.

Ginsburg and Lynch [1976] compare grammars for size complexity, in terms of numbers of grammar symbols and productions. They show that only linear improvement in size can be obtained for the CFGs. Bertsch [1983] defines a size measure of efficiency of a CFG as the sum over all productions of the lengths of righthand sides, and gives optimal forms for expressions in Pascal based on experiments measuring parsing times. Results in this area are of relevance when considering the size of a particular CFG, which determines the size of its parser and also the number of symbols or productions to be considered by an automatic error handling scheme.

Some error handling schemes require the use of special symbols in the language; a context-free language (CFL) to be handled by such a scheme must be of suitable design. Pai and Kieburtz [1979] define *fiducial* symbols of a language which are used by their error recovery algorithm as tokens which constrain the string following them in a sentential form in particular ways. Tai [1980] discusses *predictors,* which are substrings of a sentence that determine the contents of the parse stack when the symbols in the substring are parsed, and shows how these may be used in error recovery. Mauney and Fischer [1988] refine these notions to identify particular symbols, parsing ahead to which ensures that a certain class of repairs can be constructed.

3. ERROR REPORTING - THE USER INTERFACE

Good error reporting is important to the user of a language translation system. For a novice user, the error messages may make the difference between being able to run a program and obtain results, and being frustrated with a program that will not compile. An experienced user may not consider that error messages are as important as, for example, the quality of code generated. But even the most knowledgeable user makes mistakes like keyboard slips from time to time. Good error messages give the ability to correct these errors readily, and hence make a system more pleasant and efficient to use. Error reporting constitutes the user interface of an error handling scheme. As such, it is only possible to consider certain aspects of it independently from any method used to handle errors.

Most work on error handling recognizes the importance of the user interface and gives good attention to error messages. However, Sippu [1981] in his survey notes that it is curious that none of his references is solely devoted to syntax error reporting. Since then the papers of Brown [1982, 1983] are devoted solely to analyses of syntax error reporting by Pascal compilers. Other later papers surveyed here are concerned with the user interface in a context wider than that of compiling, paying attention to all computer system messages [Dwyer 1981; Shneiderman 1982; Norman 1983; Molich and Nielsen 1990]. Horning [1974] makes classic recommendations for both the content and the form of compiler messages to the user, including a classification of responses to errors. Brown [1982, 1983] criticizes error messages from the user's point of view, describing the current "state-of-the-art" as "appalling". He identifies the

"egotistic" compiler as one which presents information in its own terms rather than in those of the user, and he criticizes messages which say not what is wrong but what the compiler expected to find. On the other hand Kantorowitz and Laor [1986] consider that a list of the expected symbols is readily understood by the programmer; one of their prime requirements for an error handling system is the avoidance of inaccurate messages and the automatic generation of useful messages which are described as simple, honest and reliable. (Personal tastes differ in the evaluation of messages as "good".) Brown also requires the avoidance of inaccurate messages, and suggests that the development of integrated programming environments and the use of windowing systems should improve the quality of error messages. However, unless the designers of such systems pay particular attention to error handling and the resulting messages, there is no reason to suppose that improvement should come, simply because a windowing system or an integrated environment is being used. For example, the following source input in the Pascal programming language lacks a pair of parentheses for the procedure parameter on line 5:

```
1    program demo(output);
2    var i: integer;
3    begin
4       for i := 1 to 10 do
5          writeln i
6    end.
```

When submitted to MacPascal, an integrated Pascal programming environment for Apple Macintosh personal computers using windows and graphics, the diagnostic message, which pops up in a separate window from the input program, is

"This does not make sense as a statement"

In the input program window, the fifth line is indicated as the offending statement, and the symbol 'i' is highlighted within the line. The use of windows and graphics enables good visual identification of the parser defined error, but the content of the error message is still misleading and uninformative. A better message would suggest insertion of the missing pair of parentheses.

Dwyer [1981] uses behavioural theory to develop a user-friendly interface in which a system's response to erroneous input consists of repeating the input, indicating the erroneous symbols and displaying the correct format of the input. This model is particularized for a compiler which attempts to print the grammar production it expects to use. (This is unlikely to be helpful to most programmers.) Shneiderman [1982] makes recommendations about computer system messages in general including the development of quality control and guidelines, the use of acceptance tests, and the collection of user performance data. Norman [1983] also stresses the importance of analysing users' performance, particularly the errors that are made, to help in the design of interfaces that improve performance and reduce the number and effect of errors. He recommends that the state of the system and a list of options should be available to the user.

The most recent research is still making the same recommendations: two of Molich and Nielsen's nine principles for the design of a good human-computer dialogue are "Provide good

error messages" and "Prevent errors"[Molich and Nielsen 1990]. Good error messages are defensive, blaming the system rather than the user; precise, giving exact information about the problem; and constructive, suggesting how to proceed. The conclusions from an interesting experiment in dialogue evaluation are that designers of dialogues are insufficiently aware of the importance of design to prevent or tolerate errors, and that the principles involved are not common knowledge.

The reporting of errors and their treatment can be studied as a separate topic, with recommendations for improvements. But implementation of those improvements is impossible to achieve without changing the parser or the error recovery scheme.

4. ERROR DETECTION

A parser detects an error in its input when it has no legal move on its next input symbol. The problem of error detection (with respect to parser defined errors) can be said to be both defined and solved by parsers which possess the *correct prefix property*. Such a parser consumes only input that is a prefix of the language to be parsed; it halts in a configuration where the next input symbol concatenated with the previously consumed input does not form a prefix - that is, a correct-prefix parser halts at the parser defined error. Many parsers, including all LL and LR parsers, possess the correct prefix property; operator precedence parsers do not possess it. The correct prefix property is important for both error reporting and error recovery, because it enables the parser to output a diagnostic message and invoke an error recovery scheme at the point of error. The diagnostic message locates the exact incorrect symbol according to the parser defined error model. The error recovery scheme is invoked before consumption of the incorrect symbol, rather than several symbols later.

Although correct-prefix parsers never consume incorrect input symbols, Strong LL(1), SLR(1) and LALR(1) parsers may perform parsing actions before halting [Fischer et al 1979; Graham et al 1979]. Strong LL(1) parsers may pop the top non-terminal on the parsing stack, in an expansion by the empty production. SLR(1) and LALR(1) parsers may replace a sequence of grammar symbols on the parsing stack by a non-terminal in a reduction move, because the method of construction for the parsing tables uses less lookahead information than the method for canonical LR tables. Additionally, the tables of LR parsers may be compacted so that for a state where the only non-error move is a reduce move, this reduction will be indicated for all input symbols; these are called *default reductions*.

The term *immediate error detection property* is used for parsers which halt as soon as the parser defined error is the next symbol of the remaining input, without performing any actions. This property is stronger than the correct prefix property, and has the advantage for error recovery of leaving the parsing stack in the configuration in which the error is detected, providing more information for an error handling scheme. The canonical LL and LR parsers possess the immediate error detection property, but are not usually of practical interest. Ghezzi [1975] defines a subclass of LL(1) grammars which have parsers with the property. There are relevant differences between Strong LL(1) parsers and LL(1) parsers as constructed by the

algorithms in [Aho and Ullman 1972, pp. 345, 350]. Such LL(1) parsers possess the immediate error detection property, but have larger parse tables than Strong LL(1) parsers, which do not possess the immediate error detection property [Fischer et al 1979]. Both possess the correct prefix property. Fischer et al give a parsing algorithm for non-nullable Strong LL(1) grammars which does perform immediate error detection. The algorithm is improved so that it parses all Strong LL(1) grammars [Mauney and Fischer 1981].

5. ERROR RECOVERY SCHEMES

Error handling schemes range from those which quit as soon as an error is detected, to those which perform full error correction. Most error handling schemes lie somewhere in between and attempt some form of error recovery, rather than quitting, looping or crashing on the one hand, or attempting correction on the other. To be strict, true error correction cannot be achieved for all inputs by any scheme which does not consult the user about his or her intentions. However, the term *correction* is occasionally used with a less strict meaning, for example for a compiler that always produces a syntactically correct structure and the corresponding target code [Conway and Wilcox 1973]. Minimum distance error correction is the term used to describe schemes in which the user's intention is modelled by the minimum distance metric [Aho and Peterson 1972; Lyon 1974; Krawczyk 1980]. The error handling in these schemes is integrated with parsing, rather than invoked as a separate module when an error is detected. Some schemes perform locally minimum distance error correction, that is they produce an error repair which is a minimum distance error correction for the input over a bounded, local region [Anderson and Backhouse 1981; Mauney and Fischer 1982]. Most schemes however do not perform error correction, but use some form of error repair to effect error recovery and get the parser back on track. Approcahes to error repair may involve alterations to the input alone, known as local recovery, or alterations to both the input and the parse stack, known as regional recovery.

The history of work on syntax error handling dates from the nineteen-sixties. During that decade, production compilers were developed and their need for good quality error handling was identified and, to some extent, met. The area of error recovery was pioneered by CORC, the Cornell Computing language compiler [Conway and Maxwell 1963], its successor for a dialect of PL/I [Conway and Wilcox 1973], the DITRAN compiler [Moulton and Muller 1967], and the PL360 compiler [Wirth 1968]. These papers also supplied some important ideas for later work. The aim for CORC and its successor was to complete translation of every program submitted, whether correct or not. The PL360 compiler provided a heuristic solution to the problem of error recovery in a precedence parser. Although the quality of error recovery in these compilers was good, the methods used were ad-hoc and hand-coded. The techniques used have subsequently been developed for automation. The first paper on automatic error handling, and the only one to be published in the sixties, was due to Irons [1963]. It gives a parsing algorithm which carries all possible parses along and an error recovery method which uses the current stack, representing the syntax tree built so far, and the upcoming input to alter the input so that parsing can continue.

The seventies and eighties have seen the development of the topic of error recovery and the

publication of many papers. The next sections of this paper attempt to explain the basic techniques of error recovery schemes and to give a comprehensive survey of recent work. (Bibliographies of less recent work may be found in [Ciesinger 1979], [Sippu 1981] and [Mattern 1984].)

The primary aim of an error recovery scheme is to restore the parser, which is in a state from which it cannot proceed, to a state in which parsing of the input can be resumed. The strategy may be to use global or local information, or a combination of the two. Actions taken will usually include making alterations to the parsing configuration, that is to the input and possibly to the parse stack. Error recovery schemes can be classified into three areas, those performing global correction, those altering only the input (local recovery) and those altering the input and the parse stack (regional recovery). A scheme that performs global correction uses full information about the entire input in order to make its choices. A scheme that alters the input alone is in effect making an assumption about the state of the parsing stack, that it is in some sense correct and forms a basis from which parsing can restart. Left context information, about previously parsed input, may be used to decide on a suitable repair, but may not be changed. A disadvantage of this approach is that an error which is not detected until some symbols after the minimum distance error is impossible to correct. A scheme that alters the stack does not necessarily assume that the previously parsed input is correct, and has greater flexibility in its choice of restart state. A disadvantage of this approach is that recovery actions that pop states from the stack simulate the undoing of previous parsing actions. In a compiler these actions frequently include code generation and alterations to the symbol table, which are hard to undo in practice.

5.1 Global Correction

Schemes that perform global correction adopt an approach in which the entire input is taken into account when handling errors, with the aim of producing a minimum distance or least-cost error correction for an incorrect string. Global correction is effected by adapting a general context-free parsing algorithm for use on the entire input, incorporating error transformations either directly into the parsing algorithm or by means of special productions added to the CFG. This approach contrasts with one in which an efficient parser analyzes the input until an illegal symbol is met, at which point a separate error handling scheme is activated.

The correction of errors is typically modelled by considering three types of error transformation of a string: replacement, insertion or deletion of a single symbol. Aho and Peterson [1972] give an algorithm based on that of Earley [1970] which parses any input string according to a given CFG using the smallest possible number of transformations required to map the string into a syntactically correct string. Error productions which simulate the above transformations are added to the given CFG, yielding an ambiguous grammar generating all sentences over the input alphabet. This grammar is used to guide the parse and count the number of error transformations made. The algorithm requires $O(n^3)$ time and $O(n^2)$ space, rendering it impractical for use in a production compiler. Lyon [1974] also gives an algorithm based on

Earley's algorithm which computes the minimum number of error transformations needed to transform an incorrect string into a sentence. The algorithm handles all context-free languages.

A similar approach performs global recovery by restricting the set of errors that can be handled to those involving single edit operations on a limited set of input symbols, separators and parenthesis structure symbols [Krawczyk 1980]. A CFG which models such errors is constructed by adding to the original CFG productions with such symbols deleted or replaced. The resulting language must lie within the class of languages that can be handled by the chosen parsing method. The error correction is then performed by the parser itself for this CFG. The LR parsing method is extended to handle the ambiguity which commonly arises from the deletion of parenthesis structure symbols from productions.

5.2 Regional Recovery - Alterations to the Parsing Stack and the Input

The central problem for an error recovery scheme which is not integrated with the parser is that of determining how to transform a parser configuration in which there is no legal move into one in which there is at least one legal move, so that parsing can continue. This section will consider solutions to this problem using several different techniques, whose common element is modification of both the parsing stack and the input. The problem is now one of determining how much of the parse stack and input to alter and what the inserted stack and input symbols should be. In effect, the error recovery method must aim to identify and replace a portion or *phrase* of the original input extending to left and right of the symbol at the point of discovery of error; input symbols to the left of the error symbol have already been parsed and are thus represented in some way on the parse stack.

5.2.1 Panic Mode

One of the most basic forms of error recovery, *panic mode,* has been extensively used in production compilers and has provided a basis for many more sophisticated schemes [McKeeman et al 1970]. In panic mode, input symbols are deleted until one of a set of marker symbols, determined by the compiler-writer for the particular language, is met. The parse stack is then popped until the marker symbol can be shifted. Panic mode recovery is easy to implement and efficient in operation, but often results in the deletion of several input and stack symbols, leading to indifferent quality of error handling.

More complex schemes than panic mode use various techniques to identify the phrase to be replaced: forward moves, error productions, marker symbols, costs, and combinations of these techniques.

5.2.2 Forward Moves

The technique of modifying the parse stack by performing an alternative reduction is known as *phrase-level recovery*, due to Wirth [1968] and formalized for precedence parsers and named by Leinius [1970]. Phrase-level recovery aims to identify a phrase of the original input in which the

error is supposed to occur, and to recover by replacing that phrase with a suitable non-terminal. A *forward move* is used to gain information about the upcoming input. In the forward move, the error recovery scheme temporarily passes control back to the parser so that some of the remaining input can be parsed. After the forward move, the recovery scheme uses the information gained from the parse ahead, to find the shortest sequence of stack and input symbols, bounded by the appropriate precedence relations, that can be replaced by a unique non-terminal which gives a valid reduction and subsequent legal parser configuration. The technique of phrase-level recovery with a forward move can also be used in conjuction with an LR parser [James 1972].

A *backward move* can be used before a forward move, to obtain further information about already parsed input and the stack configuration, or to alter the stack by making further reductions [Graham and Rhodes 1975]. Grammar symbols are assigned costs for insertion and deletion operations, so that a string of symbols with least cost can be chosen as a modification to the stack. An alternative use of a backward move makes the assumption that errors occur in clusters containing up to a fixed number of errors [Levy 1975]. The backward move is used to find the position of the last special symbol met, called a *beacon*; the forward move is used to generate legal continuations and to select one.

Leinius' method of phrase-level recovery has been formalized and extended for LR parsers, and used together with spelling correction and local correction as a basis for the generation of automatic error handling in the compiler-writing system HLP78 [Sippu 1981; Sippu and Soisalon-Soininen 1982, 1983]. Experiences gained in using the HLP78 compiler writing system suggest that ad-hoc modifications to CFGs give better error recovery and diagnostics than can be provided by using the system with the most natural CFGs [Raiha 1980; Raiha et al 1983]. Experimental results show that the quality of phrase-level recovery depends heavily on the form of the productions. For example, if a list of statements in Pascal is generated by the productions

statement-sequence → *statement-sequence* ; *statement* | *statement*

then a semicolon missing from between two statements causes the deletion of the whole of the second statement. Phrase-level recovery action chooses a non-terminal as a reduction goal and isolates a phrase in the input which is to be replaced by that goal; in this example the non-terminal *statement-sequence* is chosen as the reduction goal and the phrase is delimited at its rightmost end by a semicolon or the **end** symbol. If the less natural productions

statement-sequence → *statement_list semicolon statement* | *statement*

semicolon → ;

are used instead, the non-terminal *semicolon* is chosen as the reduction goal and the phrase in the input to be replaced is the empty phrase between the two statements. In this case phrase-level recovery is equivalent to local recovery, because the form of the productions has been carefully chosen to ensure insertion or replacement of a suitable terminal.

Another scheme for LR parsers based on the ideas of Levy and Graham and Rhodes restarts the parser on a forward move and then attempts repair by insertion of a single symbol [Mickunas

and Modry 1978]. Several forward moves may be considered, as the restart states are all those parser states to which there is a transition after shifting the current input symbol.

5.2.3 Error Productions

An error production is a production added to a CFG which includes the use of a new terminal **error** in the right-hand side of the production. The **error** terminal will be incorporated into the parsing tables, possibly by a parser-generator, and used to guide error recovery. The popular LALR parser-generator *yacc* [Johnson 1975] uses such an automatic error recovery scheme, due to Aho and Johnson [1974]. The user adds error productions of the form $A \rightarrow$ **error** or $A \rightarrow$ ... **error** ... to the CFG forming the *yacc* input. The error recovery scheme pops stack states until the top state can shift the **error** symbol. The stack is then reduced by the appropriate error production, and input is discarded until a legal shift symbol is met. Three legal input symbols must be shifted before recovery is complete - any intervening illegal symbols are deleted. The same error recovery scheme is used for *Bison*, the GNU parser-generator [Donelly and Stallman 1988]. Poonen [1977] describes a similar scheme for LR parsers, which also uses an **error** terminal in productions added to the grammar by the compiler-writer. When an error is detected, all states in the stack which can shift **error** are inspected to determine all their legal shift symbols. Input is deleted until one of these is met and the stack is popped until a state which can shift this symbol is on top. This method may delete more stack states, but fewer input symbols, than that of Aho and Johnson.

5.2.4 Marker Symbols

Marker symbols are distinguished terminals of the CFG used in error recovery, usually to delimit portions of the input within which alterations may be made.

The Zurich Pascal compiler contains a top-down syntax analyzer implemented by the method of recursive descent [Wirth 1971]. The method used for error recovery is a sophisticated development of panic mode recovery employing sets of marker symbols known as *follow sets* [Wirth 1976, Amman 1978]. At the point of detection of error, the top-down parser has a goal non-terminal on top of its stack, associated with which is a follow set of terminals. The follow set contains those terminals which are legal input symbols for the current parser configuration, together with those terminals which, although not legal, should not be skipped. Error recovery consists of skipping input until such a symbol is met, simultaneously rebuilding the partial syntax tree. The recovery method is implemented by an error procedure with a formal parameter that represents a set of symbols compatible with the current syntax tree. An additional parameter may be used to represent a set of symbols compatible with a rebuilt tree [Amman 1978]. Each parser procedure must call the error procedure with the appropriate parameters. The error recovery scheme in the IBM Pascal/VS compiler is also based on the use of follow sets [Spenke et al 1984]. The method is made more systematic and suitable for generating directly (automatically) from the CFG by Hartmann [1977], with further improvements by Pemberton

[1980]. For a formal description of follow sets and algorithms for error recovery using follow sets and error productions, see Stirling [1985]. These error productions are productions which contain instances of an error message, rather than the error productions of Aho and Johnson [1974] which contain instances of an error token. Kantorowitz and Laor [1986] use a scheme for LL(1) parsers based on follow sets in which ensuring the usefulness of error messages generated is of primary importance. An error message consists of a list of legal symbols at the point of discovery of error and an indication of which input symbols are skipped in recovery. Lewi et al [1978] describe a technique for top down and LR parsers which uses marker symbols supplied by the compiler-writer to delimit the start and end of a phrase of the input, and a non-terminal reduction goal for such a phrase.

Pai and Kieburtz [1979, 1980] also use marker or *fiducial* (trustworthy) symbols to identify a phrase for recovery in an LL(1) parser. The parser scans input until such a symbol is met, when the stack is popped until a sentential suffix beginning with the fiducial symbol can be accepted. Tai [1980b] and Mauney and Fischer [1988] discuss similar models for marker symbols.

5.2.5 Costs

Costs of edit operations can be used to determine a locally least-cost repair [Graham and Rhodes 1975]. Any method can be used to construct a set of possible repairs, followed by computation of the cost of the repairs with respect to the actual input. A development of this technique employs an additional measure, the *reliability value* of an input symbol [Spenke et al 1984]. The reliability value measures the probability that the symbol was not placed in the input accidentally. Insertion and deletion operations are simulated by popping the stack and discarding input symbols until the parser is restored to a valid configuration. The choice of repair is based on a comparison of the total cost of the edit operations with the reliability of the next actual input symbol to be accepted.

5.2.6 Combinations

Combinations of the above techniques are used, particulary in those schemes which can be described as two-level. A two-level scheme has two separate stages for error recovery, the second of which is entered only if the first stage fails to produce an acceptable recovery action. Usually the idea is that a simple error may be corrected by a simple technique, but if the error is complex or cannot be repaired by the simple technique, then a more complicated technique must be brought to bear on the problem. Graham, Haley and Joy [1979] give a two-level scheme for LR parsers which uses several ideas : a forward move, to obtain right context, and costs for edit operations, to guide the choice of a repair to the input, in the first stage; and error productions for phrase-level recovery in the second stage if no first-level repair succeeds. The scheme also uses semantic information, in hand-coded recovery procedures. The use of semantic information for syntax error recovery has been formalized and extended [Corbett 1985]. A potential repair can be tested with a forward move with execution of semantic actions. This approach yields further

information to guide the choice of repair, but has the drawback that it requires the compiler to be able to undo the effects of semantic actions.

Burke and Fisher [1982] give a two-level scheme for LR parsers which attempts local recovery of a single edit operation first, backing up on the parse stack, followed if necessary by phrase-level recovery. The notion of *scope* is used to control the region in which local recovery is attempted, and to provide the goal for secondary recovery by insertion of a string which will close scope. This scheme is extended to one with three levels of recovery [Burke and Fisher 1987]. The first level performs simple recovery as before; the second level performs scope recovery which deletes and/or inserts input symbols to close an open scope; the third level performs phrase-level recovery which deletes input before or after the symbol at the point of detection of error. Boullier and Jourdan [1987] give a two-level scheme for table-driven parsers and lexical analyzers. The first level attempts insertions, deletions and replacements at the point of detection of error with the aim of finding a match in a list of corrections. The second stage deletes input up to a *key terminal* or marker symbol, and pops the stack until that symbol can be accepted. A similar two-level scheme for LR parsers also attempts a single repair at the error point, but uses a forward move on the input to determine whether the repair succeeds [Dain 1989]. The second level chooses a non-terminal from the parse table as the reduction goal for a phrase-level recovery which pops the stack and deletes input.

5.3 Local Recovery - Alterations to the Input Alone

The designer of an error recovery scheme may choose to limit the possible transformations of parser configurations by allowing modifications to the unexpended input only. One of the reasons for this choice is that modifying the stack may require changing semantic information associated with the parsed input, which is difficult to achieve in practice [Gries 1971]. The problems to be solved are firstly, determining how much of the unexpended input, from the symbol at the point of discovery of error and possibly extending rightwards, should be altered, and secondly, determining an appropriate replacement string or repair. Input symbols to the left of the error symbol have already been parsed and are not eligible for edit operations; neither is the current state of the parser to be altered. Several of the techniques used to identify symbols to be replaced in schemes which alter both stack and input are also used here, namely forward moves, marker symbols, and costs. The use of error productions is inappropriate, as this technique involves replacing stack and input symbols by a suitable non-terminal.

5.3.1 Forward Move

A forward move on the input can be used to obtain information on how to proceed without altering the parse stack. An SLR parser can be extended with extra states to handle errors with a forward move [Druseikis and Ripley 1976]. At the point of detection of error, a special non-SLR parser is activated to parse the upcoming input as far as possible. The initial state of this parser is formed from all those states which can accept the error symbol, called recovery states. When this forward move terminates, the original SLR parser is restarted, using

information gained from the forward move to choose a restart state which allows parsing of the actual input to continue. This is purely recovery; no actual repair to the input is constructed. However, the method can simulate the effect of deleting several symbols from the input. A similar method for LR parsers develops the forward move as a parallel parse by the extra recovery states, rather than a union of recovery states [Pennello and DeRemer 1978]. An attempt is made to repair the input by insertion, deletion or replacement of a single input symbol, using information gained from the forward move. LR parsers generated by the HLP84 processor generator also use a forward move by a parser augmented with recovery states [Koskimies et al 1988b]. An initial recovery state is constructed from all parser states that have a transition on any of a fixed set of "safe" symbols. On detection of an error, the entire parse stack is deleted and the parser restarts from the initial recovery state. A method for LR parsers uses a forward move to generate possible repair strings, finally choosing the repair which is closest in minimum distance to the actual input [Dain 1989].

5.3.2 Marker Symbols

Marker symbols are typically used to delimit a portion of the input in which alterations may be made. A prior choice of marker symbols limits the repair to a local correction. Transformations to the error symbol are tried first and if these do not enable the parser to continue, the input is backed up a symbol at a time [Feycock and Lazarus 1976]. Turner [1977] develops a variant of panic mode recovery for recursive descent parsers in which input symbols are skipped until a marker symbol is encountered. Marker symbols can be determined automatically as those symbols for which the current parser procedure is checking; additionally the compiler-writer can hand-tune procedures with extra marker symbols. Röhrich [1980] gives methods for automatic error recovery in LL and LR parsers by insertion and deletion operations only on the input. A valid continuation string is found and input is deleted until an *anchor* symbol is met, that is any input symbol which is contained in the continuation. The appropriate prefix of the continuation string is inserted and parsing continues from the anchor symbol. Although the method is automatic, Röhrich also uses semantic considerations applied to common properties of programming languages to hand-tune the choice of anchor symbols, in order to decrease the probability of what he calls avalanches of spurious errors. A similar approach is taken by Chytil and Demner [1987], differing from that of Röhrich in that input is deleted until the first symbol from a fixed set of *skeletal* symbols is met. Informally, skeletal symbols are chosen so that they delimit the region in which an error can be corrected. The authors note that finding a skeletal set for a programming language may be difficult and show that it is undecidable for a CFL whether a set of symbols is its skeletal set. Barnard and Holt [1981] also use *synchronizing* symbols to mark phrases in the input.

5.3.3 Costs

Costs of edit operations, symbols or their equivalent are used to choose one from a set of potential repairs. Tai [1978] describes a technique called pattern mapping which is used to

choose a local repair to the input. Patterns model the transformation of one string into another and have associated costs. A list of patterns is scanned for a successful match with the upcoming input. The method is implemented for an SLR(1) parser. Tai [1980a] also develops a technique for LL(1) parsers which uses costs of edit operations to choose a locally minimum distance correction, based on a formal model which assumes that errors occur in clusters. Backhouse [1979] describes a method for choosing a local repair to the input based on costs, and applies it in a recursive descent parser. Anderson et al [1983] assess an implementation of the method and Backhouse [1984] analyses the data flow problems arising from optimization of the parser for practical purposes. Anderson and Backhouse [1981] incorporate the same error handling method into Earley's parsing algorithm. Fischer et al [1980] use costs to choose a repair in a scheme for LL(1) parsers which uses only the insert operation. They show that most common programming languages can be modified so that they are contained in the subclass of LL(1) grammars which can be parsed and corrected by this method. A similar method for LR(1) parsers is developed by Dion [1982], but the technique is less suitable for LR parsing as the stack does not directly contain the information that guides the choice of repair.

Mauney and Fischer [1982] extend the general CFL parser of Graham et al [1980] to perform repair which is least-cost over some region of the input. The parsing algorithm simulates the edit operations of insertion, deletion and replacement and includes a computation of costs. It is only invoked when an error is detected by the usual (LL or LR) parser; after some region has been parsed, the chosen repair can be inserted on the input and control returned.

6. CRITERIA FOR ERROR RECOVERY

In this section a framework in which to evaluate error recovery schemes is established. An error recovery scheme should be assessed on several points: a good scheme is one which is satisfactory for all those points which the user considers important. Within the context of language translation systems, a compiler should never loop or crash and it should attempt to detect and report as many errors as possible [Horning 1974]. Six different types of error handling action are identified by Horning, in increasing order of desirability being to crash or loop, to produce invalid output, to quit, to recover and continue checking, to repair and continue compilation, to correct. Röhrich [1980] requires an error handling scheme to detect and report all syntactic errors; to emit for each error a clear message which describes the nature and location of the error and explains the recovery action taken; and to recover and continue parsing. In addition, the scheme should neither enlarge the parser's interface to other compiler modules substantially, nor affect adversely the parsing of correct input. Spenke et al [1984] state desirable properties for an error handling scheme to be used in a compiler-generator which include language independence, efficiency, automatic generation of user-oriented messages, detection of all errors, and introduction of no spurious errors. They note that the last two properties cannot be completely formalized.

Goals will be set for an error recovery scheme in the three general areas of performance, efficiency and ease of use. Performance is concerned with the quality of detection and reporting

of errors; efficiency is concerned with space and time requirements; ease of use is concerned with effort by the compiler writer. There are likely to be trade-offs in these areas, particularly between performance and efficiency.

6.1 Performance

Considering the area of performance first, minimum distance error correction seems to be the best goal: it comes closest, of all models, to the concept of altering input to the programmer's intention, and it is completely formalized. However, the best known algorithms have time complexity $O(n^3)$ and space complexity $O(n^2)$ and hence are not sufficiently practical in time and space for production compilers [Aho and Peterson 1972; Anderson and Backhouse 1981; Mauney and Fischer 1982]. The minimum distance model can be used as a theoretical yardstick against which to measure the performance of practical ($O(n)$) methods.

Programmers normally wish to make their programs syntactically correct as quickly as possible, that is with as few passes through the compiler as possible. The parser should therefore attempt to detect all syntactic errors in the input. This implies that the parser should analyse all the input, so that no errors go undetected in deleted portions of the input. Thus there must be some means of restarting the parser from an error configuration. In the case of a simple error which can be corrected by a single edit operation, the parser can be restored to a legal configuration by making that edit operation on the input. This repair will frequently be the most natural in the sense that it appears to be close to the programmer's intention, even though there may be a choice of repair. Deleting or replacing one or more input symbols, rather than inserting symbols, may conflict with the aim of parsing as much as possible of the input, but is often a more natural recovery action. The following program from the Ripley collection of Pascal programs [Ripley and Druseikis 1978] gives an example of an error where the most natural repair is a deletion.

```
1    program p(input, output);
2    begin  if a < b ;
3       then x := 1
4    end.
```

The semicolon on line 2, which will be detected as an illegal symbol by any correct prefix parser, should be deleted. A correction by insertion only requires the insertion of a statement before the semicolon, precipitating a further error at the symbol 'then' requiring further insertions, indicated in italics:

```
1    program p(input, output);
2    begin  if a < b then x := 1 ;
3       if boolean then x := 1
4    end.
```

In the case of a more complex error which cannot be corrected by a single edit operation, there are many ways of restoring the parser to a legal configuration, in different combinations of alterations to the input and alterations to the parsing stack. The parser should be put back on

track, in the sense that it will proceed to parse some or all of the remaining input.

The parser can assist programmers in correction of errors by producing error messages that give the place in the input at which an error is detected and state what recovery action is taken. Messages should be directed towards the user, describing the error in terms of what the user has done rather than what happened in the parser, and they should be expressed in natural language and the source language, rather than in numbers which have to be looked up in a table, grammar symbols, internal representation or target language [Horning 1974; Shneiderman 1982; Brown 1982, 1983].

6.2 Efficiency

The second area to be considered is that of efficiency. The parser must remain usable: the space requirements of the error handling scheme should be satisfiable and the time requirements should be acceptable to the user. There should be no additional time requirements for parsing correct input. Users will accept some delay in parsing incorrect input providing that parsing correct input is not delayed and that the error handling scheme provides them with information which helps them to correct their errors. Theoretical measures of time complexity have some use in evaluating acceptability, and ideally an error handling scheme should have the same complexity as the parser, but if the constants involved are large these measures may not be of much practical significance.

6.3 Ease of Use

The third area is ease of use by the compiler writer. It is not sufficient to design an efficient error recovery scheme with good performance, if such a scheme is difficult to incorporate into a compiler. The method of choice for producing a parser is to use a parser-generator which produces efficient parsers for a wide class of programming languages: LR parsers are efficient and define the DCFLS, but are difficult to produce by hand and are frequently produced by a software tool such as the LALR(1) parser-generator *yacc* [Johnson 1975]. It follows that an error recovery scheme should also be produced by a software tool. If a scheme is to be incorporated into a parser-generator, it must be capable of automatic generation and independent of source language. To make it capable of completely automatic generation, a compiler writer should not need to understand the error handling scheme in order to use the parser-generator. In particular, if the scheme is to be incorporated into an existing parser-generator, no specifications additional to the existing ones should be needed as input to the parser-generator.

6.4 Goals

The goals for an error handling scheme can be summarized informally as follows:

(1) to detect all errors in the input.

(2) to parse as much as possible of the input.

(3) to repair simple errors and recover from complex errors.

(4) to generate good error messages.

(5) to have practical requirements in time and space.
(6) to have no effect on the analysis of correct input.
(7) to be capable of automatic generation.
(8) to be capable of incorporation into a practical parser-generator.

7. EVALUATION OF ERROR RECOVERY SCHEMES

What advances have been made since 1980, when Aho remarked that automatic generation of good diagnostics and error recovery methods was still lacking in translator writing systems? Many schemes have been developed since 1980. These will be considered with respect to quality of performance, error messages, capability for automatic generation, and practicality.

7.1 Quantitative Performance Evaluation and Comparison

The question of quality can only be answered by analysing the performance of schemes in practice on faulty source programs. There is no theoretical basis for judging whether a scheme performs well. For that reason, this section is confined to those schemes which have been implemented and whose performance has been evaluated.

A method of evaluation used by many authors is to construct a syntax analyser, with error recovery, for Pascal, and to test it on the suite of student programs collected by Ripley and Druseikis [1978]. The error recovery achieved is then evaluated according to various criteria. The criteria generally take into account error messages as well as action taken to recover. Sippu and Soisalon-Soininen [1983] count the number of missed errors, the number of reports of non-existent errors, and the number of tokens skipped in recovery actions. They also classify a recovery action as *excellent* if it is the same as a "competent programmer" might take, *good* if it is not excellent but has no undesirable side effects, *fair* if it induces one missed error or one non-existent error, and *poor* otherwise. Other authors use similar criteria, classifying Sippu's *fair* and *poor* actions together as *poor* [Pai and Kieburtz 1978; Pennello and DeRemer 1978; Burke and Fisher 1982; Spenke et al 1984; Boullier and Jourdan 1987]. Stirling [1985] assesses three schemes based on follow sets using slightly different evaluation criteria: a recovery action is *good* if it produces "the most plausible repair", *marginal* if it produces either a minimum distance correction which is not the most plausible repair, or a repair which is neither minimum distance nor plausible but has no effect upon the parsing of subsequent symbols, and *poor* otherwise. Anderson et al [1983] use the same criteria as Stirling to assess the scheme of Anderson and Backhouse [1981]. Dain [1989] uses a formal measure, the difference between the number of minimum distance errors and the number of edit operations made by the recovery method. If this measure is 0 then optimum recovery has been made.

For the purposes of comparison, we classify as *acceptable* both the excellent and good repairs of Sippu, the repairs of measure 0 and 1 of Dain, and the good and marginal repairs of Stirling (although Stirling remarks that marginal repairs are not generally acceptable, because they represent inaccurately diagnosed errors; we judge them to be comparable with the good actions of Sippu because none of these actions have undesirable side effects). Fig. 1 shows the

percentages of recovery actions performed by the schemes on the Ripley suite that are thus classified as acceptable. The schemes are identified by the names of the author[s] of the scheme. The reference following gives the paper in which the scheme is presented, except in the case of Wirth's follow set scheme and the IBM Pascal/VS scheme, in which cases the reference is to the paper citing the results of assessment.

Dain (Recovery Method 1) [1989]	57%
Stirling (folset system) [1985]	66%
Stirling (folfolset system) [1985]	66%
Sippu and Soisalon-Soininen [1983]	67%
Pennello and DeRemer [1978]	70%
Wirth [Stirling 1985]	72%
Boullier and Jourdan [1987]	75%
IBM Pascal/VS [Spenke et al 1984]	77%
Pai and Kieburtz [1978]	77%
Anderson and Backhouse [1981]	79%
Dain (Recovery Method 2) [1989]	84%
Spenke et al [1984]	91%
Burke and Fisher [1982]	98%

Fig. 1. Percentage of recovery actions which are acceptable

In Fig. 1, most of the schemes give acceptable recovery at least three-quarters of the time. This would seem to provide a minimum standard for performance, which is both acceptable to the user and achievable in practice. The very high percentages reported by Spenke et al and Burke and Fisher may be partly due to hand-tuning of their recovery schemes for Pascal. (Both these schemes employ language-specific data on input symbols.) According to Ripley and Druseikis' analysis of their suite of Pascal programs, 88% of the syntax errors are simple, single token errors (41% missing tokens, 39% incorrect tokens, 8% extra tokens) which could be corrected by a single edit operation (41% insertions, 39% replacements, 8% deletions). It would not be unreasonable to expect a recovery scheme to handle most cases of simple errors well, and this may be one reason why most of the schemes mentioned above achieve similar standards. It is also possible that schemes which achieve lower standards are not considered to be worth publishing.

Disadvantages of using the Ripley collection of Pascal programs are firstly, the program texts have been edited and shortened, so that errors occur more frequently than happens in actual practice. This is a disadvantage for most recovery schemes, which use upcoming input to guide the choice of repair. Secondly, and less importantly, the collection is from a select user community, students at a university, and thus may not accurately represent other forms of user community. Thirdly, the evaluation of recovery schemes is limited to their performance on

Pascal. The advantages are firstly, that many authors have used the collection, and it is thus possible to make objective comparisons between different schemes. Secondly, the collection has been carefully made, and analyzed for the differing kinds of syntax error. There is an obvious need for further collections of programs, for different languages and different user communities, to be made and accepted as benchmarks in the same way as the Ripley collection.

Some schemes are not assessed by the above method, but their performance is demonstrated on one or two example programs. These are not usually representative of the majority of programs. The example Algol program of Graham and Rhodes [1975], which has been used by other authors [Druseikis and Ripley 1976; Tai 1978; Graham et al 1979; Fischer et al 1980] as a demonstration of their schemes, contains several artificial errors. The general performance of a scheme cannot fairly be judged from its performance on a single example of this kind.

7.2 Error Messages

In this section we consider the error messages issued by the schemes evaluated above, where possible. We are interested in the format and wording of messages, rather than the quality of error recovery in the examples. The authors Pennello and DeRemer [1978]; Burke and Fisher [1982]; Sippu and Soisalon-Soininen [1983]; Spenke et al [1984]; Stirling [1985]; Bouillier and Jordan [1987]; and Dain [1989] all give examples of error messages issued by their schemes. (There are no examples given by Pai and Kieburtz [1978] or Anderson and Backhouse [1981].) For an error message to be precise and constructive, four properties are desirable: it should locate errors accurately in the source text, it should explain the nature of the error, it should tell the repair made, and it should not use technical terms such as non-terminal names. Fig. 2 provides a summary of error messages that shows which of the schemes possess these properties. Perhaps most importantly for the user, all the schemes locate errors accurately. All but the scheme of Burke and Fisher give the effective repair made to the input.

	Locates error in source	Tells nature of error	Gives repair made	No use of technical terms
Pennello and DeRemer [1978]	•		•	
Burke and Fisher [1982]	•	•		
Sippu and Soisalon-Soininen [1983]	•	•	•	
Spenke et al [1984]	•		•	
Stirling [1985]	•		•	•
Boullier and Jourdan [1987]	•		•	
Dain (Recovery Method 1) [1989]	•		•	
Dain (Recovery Method 2) [1989]	•		•	•

Fig. 2. Desirable properties of error messages

Each message of Pennello and DeRemer extends to four lines, of which only the first and last are helpful to the user:

```
ERROR    Line 4, token 4, unexpected "+"
FOLLOWS  Block_body "I"
FORWARD  ? Primary ? Relationop Expr  ? then  ? go   (ERROR)
REPAIR     "if" was inserted after Blockbody and before "I".
```

The second and third lines provide too much information, expressed in technical terms. Burke and Fisher synthesize messages from the recovery action, but do not give the exact repair made. It is hinted at in this example:

```
*** Syntax Error: "=" expected instead of ":="
```

but not in this one:

```
      for i:=1  step 1 until listsize -1 do
       <---------^^^^----------------------->
*** Syntax Error: Bad statement
```

Here there is little help to the user on how to proceed; also the use of the word "Bad" is unnecessarily pejorative. Sippu and Soisalon-Soininen pay special attention to providing two-part messages whose first part explains the perceived nature of the problem. The recovery action taken is then given in the second part:

```
'(' expected.
The recovery action was to insert '('.
```

The first part can be very helpful to the user. The second part is helpful where good recovery is made, but less so when a complex phrase-level recovery is necessary, as the use of non-terminal names to describe the repair can be confusing:

```
No Statement can start with this.
The recovery action was to replace 'TYPE' Type-Definition-List
';' 'BEGIN' Statement-List ';' by Routine-Body.
```

The schemes of Spenke et al, Stirling and Dain all give messages in simple format, synthesized from the recovery action taken. Examples from Spenke et al are, for simple recovery:

```
';' deleted before end of line
```

and for more complex recovery:

```
'until FR' changed to 'to <EXPRESSION>'
```

Stirling's messages either take the form "`Symbol deleted`" or "*x* `inserted`". Complex recovery requires repetition of these formats. Examples from Dain are, for simple recovery:

```
';' deleted
```

For more complex recovery, Recovery Method 1 uses grammar names for non-terminals:

```
'for i := 1 step 1 until listsize - 1 do
   x := 1' replaced by 'statement'
```

whereas Recovery Method 2 uses the source text and tokens only:

```
for i := 1 step 1 until listsize - 1 do
-----------^ replace 'step' with 'to'
------------------^ replace 'until' with '&'
```

The grammar writer must supply the messages for the scheme of Boullier and Jordan. The examples they give of messages written for their Pascal parser are mainly clear and helpful, except for the following:

```
**** Warning  (0) : ":=" is inserted before "+".
**** Error  (1) : Global recovery.
**** Warning (1) : Parsing resumes on "then".
**** Warning (2) : "to" is inserted before "L1".
**** Warning (1) : "is" is replaced by ":=".
```

This message may be criticized on the grounds that it is long, it uses the words "Warning" and "Error" too much, unexplained numbers are attached, and unfamiliar technical terms are used on the second and third lines.

Personal taste plays a part in the assessment of error messages and readers may make their own judgements on the above examples. The author's preference is for messages which attempt to explain the nature of the problem, as well as giving the effective repair. This is difficult to do in practice and only the scheme of Sippu and Soisalon-Soininen attempts it for all cases. The scheme would be even better if it did not use technical terms. Unfortunately the use of non-terminal names cannot be avoided altogether, as the scheme uses phrase-level recovery, but less frequent use could be made: the actual source text replaced could be given, rather than the syntactic structure.

7.3 Automatic Generation

From the compiler-writer's point of view, it is very convenient to use an error recovery scheme that can be generated automatically, particularly if the parser is to be generated automatically. Several of the schemes whose performance has been evaluated above are suitable for automatic generation. This section considers those schemes that have actually been incorporated into automatic parser-generators [Röhrich 1980; Fischer et al 1980; Anderson and Backhouse 1981; Sippu and Soisalon-Soininen 1983; Spenke et al 1984; Boullier and Jourdan 1987; Gray 1987; Koskimies et al 1988b; Grune and Jacobs 1988; Dain 1989].

Some of these schemes cannot be said to be completely automatically generated, because they require the compiler writer either to have some knowledge of the particular scheme, or to supply some language-specific information as additional input to the parser-generator. Sippu's scheme requires a suitably tailored CFG; Koskimies et al [1988a] report that the user of Sippu's scheme needs a thorough understanding of the error recovery method in order to write a CFG with good error recovery properties. Boullier's scheme requires error messages, correction models and optionally, lists of key terminals and terminals which should not be deleted or inserted. Koskimies' scheme requires a set of safe symbols. Spenke's scheme requires insertion costs and reliability measures. The schemes of Fischer et al and Anderson et al require

edit costs.

The schemes of Grune and Jacobs, Fischer et al, and Gray limit the class of languages which can be handled to the LL(1) CFLs. There is no accessible published performance data for the schemes of Grune and Jacobs, Gray, and Röhrich. The PGS compiler-writing system [Dencker 1985], in which Röhrich's scheme is used, is reported by Gray as producing parsers that are too slow for production compilers. Gray's own parser-generator DEER, which also incorporates a method based on Röhrich, is carefully tuned to produce parsers which are fast enough. The schemes of Dain are incorporated into the parser-generator *yacc* which is rather large and slow by today's standards.

7.3.1 The parser-generators yacc and Bison

Parsers generated automatically by *yacc* and *Bison*, as distributed with UNIX systems and by the Free Software Foundation respectively, incorporate the error recovery scheme of Aho and Johnson [1974]. This scheme has not been evaluated quantitatively using the above method, i.e. with respect to the performance of a parser for the Pascal programming language on the Ripley suite. There are difficulties in making such an evaluation because the performance varies enormously with the error productions added. However, because *yacc* and *Bison* are popular tools, widely available and widely used, particularly in the research community for developing new language translators, an attempt is made here to evaluate the scheme without experimental data. Advantages of the scheme are that it is efficient in operation, it requires no extra space, and it is straightforward to implement. There are several disadvantages. Firstly, if the compiler-writer adds no error productions to the input CFG, then no error recovery is made by the resulting parser, which will simply report the first error it detects and quit. Secondly, adding error productions often renders the CFG ambiguous, making additional requirements of the parser. But considerable understanding of the error recovery mechanism is needed in order to understand what effects error productions will have on error handling. Thirdly, deletion of the input, as with most schemes based on panic mode, occurs silently, giving the end user no indication as to how much input has been ignored. Fourthly, error messages are not automatically synthesized from the recovery action taken, and are either of the form "`Line X: syntax error`", or have to be hand-written by the compiler-writer. There is no information, for example on what might be a legal symbol, available for this purpose, and it is not possible for the compiler-writer to satisfy the accepted criteria for good error messages.

An example of the use of *yacc* and the resulting shortcomings of its error recovery scheme is given by the parser for the portable C compiler [Johnson 1978]. There are eight error productions in the *yacc* input describing the C language for this parser. Their effect on error recovery is as follows: if an error is detected in a C statement, input is skipped up to the next following semi-colon or } symbol; if an error is detected in a declaration, input is skipped up to the next semi-colon; if an error is detected in any other construct, input is skipped up to any symbol that can legally follow that construct. This constitutes a fairly crude form of panic mode recovery. The usual kind of error message issued by the parser is

```
"file.c", Line X: syntax error.
```

The message locates the syntax error but not precisely. It gives no information about the exact cause of the problem or about how to proceed. No messages are given about deleted input.

7.4 Conclusions

It is a value judgement whether a scheme that performs "excellent" or "good" recovery action three-quarters of the time is a good error recovery method; and there are other factors to be taken into consideration when making such a judgement. The end user of the scheme, the programmer, may take into consideration such things as the circumstances in which the scheme is used (for example, whether it is in a traditional batch-mode compiler or a syntax-directed editor), the efficiency of the scheme (particularly in response time), and the relative performances of available schemes. The compiler writer may also take into consideration how easily the scheme can be implemented if the parser is to be coded by hand, or how easily the specifications can be written if a parser-generator is to be used. Some compiler writers may wish to tune a scheme for a particular language; others may prefer a scheme which is language independent and requires no understanding of its method in order to be used.

Although many of the error handling schemes show good performance results and give good error messages, none of them appears to satisfy all the criteria established in Section 6. The main sticking point is the test for automatic generation, where several schemes fail because they require considerable knowledge on the compiler writer's part in order to supply the language-specific data necessary to achieve the good results. Unfortunately none of the schemes evaluated here has achieved widespread popularity or general use by production compiler writers. The end-user, the programmer, would probably agree with Aho that good diagnostics are still lacking in today's compilers.

7. FUTURE DIRECTIONS

The programming environments of the last two decades have included tools such as general-purpose text editors, non-interactive compilers and interpreters. There has been a genuine need for good syntax error handling in all these tools. Advances in hardware technology have given rise to a new generation of machines incorporating RISC chips, parallel architectures, high bandwidth communication and cheap memory. These advances are paralleled by theoretical and practical developments in software, giving rise to a new generation of programming environments. The paradigm of object-oriented design and programming has led to powerful new languages such as C++ and Smalltalk. IPSEs and CASE tools incorporate use of formal methods and structured methods in interactive graphics interfaces. More power has been placed on the programmer's desk, with more sophisticated software tools to exploit that power. In the programming environments of the future, error handling will be as important as it is today, and work is needed on the design of appropriate error handling.

Aho, Sethi and Ullman [1986, p. 164] write that "with the increasing emphasis on interactive computing and good programming environments, the trend seems to be toward simple

error-recovery mechanisms"; but we argue that there is still a need for sophisticated error-recovery mechanisms. Interactive programming still requires parsing and translation of the user's input. One of the main differences between interactive programming systems and batch compilers is that the unit of interaction between user and computer is smaller, typically an expression or function rather than a program unit. Fig. 3 gives several examples of use of an existing interactive environment for a functional programming language, Standard ML [Harper et al 1988]. In the figure the user's input is shown in italics. There are two system prompts, a primary prompt '-' and a secondary prompt '=' to indicate that further input is expected. The error messages output by the system do not meet accepted criteria: they are not clear or informative, they do not succeed in directing the user's attention to the point of error, and they are not oriented towards the user but are expressed in terms of the internal parsing mechanisms.

```
- +1;
Parse error: Was expecting ";"
In: ... < ? > +
- 5+;
Parse error: Was expecting an Expression
In: ... 5 + < ? > ;
- fun fact (x)
=     if x < 1 then 1
=     else (fact(x-1) * x;
= ;
Parse error: Insufficient repetition
In: ... 1 then 1 else ( fact ( x - 1 ) * x ; < ? > ;
```

Fig. 3. An interactive session with Standard ML.

In the design of recent parsing algorithms, scant attention has been paid to error recovery. A "lazy" recursive descent parser, composed from separate modules for each non-terminal in the CFG, has the benefit that the implementation can be in small components which are easy to modify and re-use [Koskimies 1990]. Each module only needs to know the production rules for a single non-terminal. But error recovery mechanisms are not considered, with the result that the parser will be user-unfriendly, particularly as the difficulties of recovering from a syntax error will be compounded by the lack of information that each module has about the rest of the CFG. An extension of LR parsing to handle CFGs with restrictions is implemented by extending the *yacc* and *Bison* LALR parser-generators [McKenzie 1990]. The existing error recovery mechanisms are used, but these are outdated, hard for the compiler-writer to use effectively, and they give poor performance in practice. A new parser-generator builds LR parsers that are not table-driven but execute code directly, resulting in faster run times for parsing [Horspool and

Whitney 1990]. There is no support in the parser-generator for error recovery. The use of Rörich's method is suggested, to construct a valid continuation sequence for the state of the parser that detects error. It is not explained how the sequence would be constructed; previous similar techniques use information contained in the parsing tables and presumably this information would have to be derived directly from the executable code. Another unsolved problem is how to restart, as in general some of the up-coming input will have to be deleted.

The user would benefit from further work on error handling applied in the design and development of integrated programming environments. Program development usually consists of a cycle of editing, compiling, executing and debugging, traditionally achieved with separate utilities such as a general-purpose text editor, a compiler, a loader and a debugger, each with their own user interface. Integrated programming environments are intended to support program development through a unified user interface and often include a structure or context-sensitive (language-specific) editor. This model of program development uses the paradigm of immediate computation; the editor analyses and translates the source code as it is entered, handling syntactic and static semantic errors. Whatever technique for parsing is used, there is a requirement for good interactive error handling which can be invoked by the editor.

New tools such as the Synthesizer Generator are being developed in order to assist in the design and production of new programming environments [Reps and Teitelbaum 1988]. These tools are typically used to build syntax and semantics analysers and attribute evaluators from language specifications based on the concept of an attribute grammar [Knuth 1968]. The Synthesizer Generator itself uses *yacc* to build the parser module of the editors it constructs. The resulting syntax error handling is simple: when an error is detected by the parser, the message `"Syntax error"` is displayed and the cursor is located at the last character of the incorrect token, i.e. the parser defined error. No indication of the nature of the error is given and no attempt at recovery or repair is made.

Another approach to the production of integrated programming environments makes use of existing tools such as compilers and editors, incorporating an extra layer in order to achieve a consistent user interface. For example, FIELD is an integrated environment based on UNIX tools running on top of the X11 windowing system [Reiss 1990]. Much attention is paid to the design of the messaging system, to obtain consistency and completeness. But the use of existing compilers, designed for use in a batch environment, means that the quality of messages about syntax errors is not improved. Reiss suggests use of a fast syntax checker when a syntax error is detected - and the quality of error recovery in such a tool depends on the recovery techniques used by that tool.

More sophisticated error handling should provide better recovery and better error messages. In particular, solutions are needed to the problem of how to choose an appropriate repair when there is little upcoming input to be used in making that choice. Work is also needed on a separate but related issue, the design of appropriate user interfaces for the interactive handling of errors.

REFERENCES

AHO, A. V. 1980. Translator writing systems: where do they now stand? *Computer 13*, 8, 9-14.

AHO, A. V. AND JOHNSON, S. C. 1974. LR parsing. *ACM Comput. Surv. 6*, 2, 99-124.

AHO, A. V., AND PETERSON, T. G. 1972. A minimum-distance error-correcting parser for context-free languages. *SIAM J. Comput 1*, 4, 305-312.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers - Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA.

AHO, A. V., AND ULLMAN, J. D. 1972. *The Theory of Parsing, Translating and Compiling*. Vol. I: *Parsing*. Prentice-Hall, Englewood, NJ.

AMMANN, U. 1978. Error recovery in recursive descent parsers. In *State of the Art and Future Trends in Compilation*, INRIA, Paris, 231-238.

ANDERSON, S. O., AND BACKHOUSE, R. C. 1981. Locally least-cost error recovery in Earley's algorithm. *ACM Trans. Program. Lang. Syst. 3*, 3, 318-347.

ANDERSON, S. O., BACKHOUSE, R. C., BUGGE, E. H., AND STIRLING, C. P. 1983. An assessment of locally least-cost error recovery. *Comput. J. 26*, 1, 15-24.

BACKHOUSE, R. C. 1979. *Syntax of Programming Languages, Theory and Practice*. Prentice-Hall, London.

BACKHOUSE, R. C. 1984. Global data flow analysis problems arising in locally least-cost error recovery. *ACM Trans. Program. Lang. Syst. 6*, 2, 192-214.

BARNARD, D. T., AND HOLT, R. C. 1982. Hierarchic syntax error repair for LR grammars. *Int. J. Comput. Inf. Sci. 11*, 4, 231-258.

BOUILLIER, P., AND JOURDAN, M. 1987. A new error repair and recovery scheme for lexical and syntactic analysis. *Sci. Comput. Program. 9*, 3, 271-286.

BROWN, P. J. 1982. "My system gives excellent error messages" - or does it. *Softw. Pract. Exper. 12*, 1, 91-94.

BROWN, P. J. 1983. Error messages: the neglected area of the man/machine interface? *Commun. ACM 26*, 4, 246-249.

BURKE, M. G., AND FISHER, G. A. 1982. A practical method for syntactic error diagnosis and recovery. *SIGPLAN Notices 17*, 6, 67-78.

BURKE, M. G., AND FISHER, G. A. 1987. A practical method for LR and LL syntactic error diagnosis and recovery. *ACM Trans. Program. Lang. Syst. 9*, 2, 164-197.

CHYTIL, M. P., AND DEMNER, J. 1987. Panic mode without panic. *Proc. 14th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science Vol. 267 (Ed. Ottmann), Springer-Verlag, Berlin, 260-268.

CIESINGER, J. 1979. A bibliography of error-handling. *SIGPLAN Notices 14*, 1, 16-26.

CONWAY, R. W., AND MAXWELL, W. L. 1963. CORC - the Cornell Computing Language. *Commun. ACM 6*, 6, 317-321.

CONWAY, R. W., AND WILCOX, T. R. 1973. Design and implementation of a diagnostic compiler for PL/I. *Commun. ACM 16*, 3, 169-179.

COOPER, D. 1983. *Standard Pascal User Reference Manual.* Norton, New York.

CORBETT, R. P. 1985. Static semantics and compiler error recovery. Report No. UCB/CSD 85/251, Computer Science Division (EECS), University of California, Berkeley, CA.

DAIN, J.A. 1989. Automatic Error Recovery for LR Parsers in Theory and Practice. Ph. D. thesis, University of Warwick, Coventry, UK.

DENCKER, P. 1985. Some topics in parser generation. IR-105, Vrije Universiteit, Amsterdam.

DION, B. A. 1982. Locally least-cost error correctors for context-free and context-sensitive parsers. UMI Research Press, Ann Arbor.

DONNELLY, C., AND STALLMAN, R.M. 1988. Bison - the yacc-compatible parser generator. Free Software Foundation Inc., Cambridge, MA.

DRUSEIKIS, F. C., AND RIPLEY, G. D. 1976. Error recovery for simple LR(k) parsers. *Proc. ACM Annual Conf.*, Houston, 396-400.

DWYER, B. 1981. A user-friendly algorithm. *Commun. ACM 24*, 9, 556-561.

EARLEY, J. 1970. An efficient context-free parsing algorithm. *Commun. ACM 13*, 2, 94-102.

FEYCOCK, S., AND LAZARUS, P. 1976. Syntax-directed correction of syntax errors. *Softw. Pract. Exper. 6*, 2, 207-219.

FISCHER, C. N., AND MAUNEY, J. 1980. On the role of error productions in syntactic error correction. *Computer Languages 5*, 131-139.

FISCHER, C. N., MILTON, D. R., AND QUIRING, S. B. 1980. Efficient LL(1) error correction and recovery using only insertions. *Acta Inf. 13*, 141-154.

FISCHER, C. N., TAI, K. C., AND MILTON, D. R. 1979. Immediate error detection in strong LL(1) parsers. *Inf. Process. Lett. 8*, 261-266.

GRAHAM, S. L., HARRISON, M. A., AND RUZZO, W. L. 1980. An improved context-free recognizer. *ACM Trans. Program. Lang. Syst. 2*, 3, 415-462.

GRAHAM, S. L., HALEY, C. B., AND JOY, W. N. 1979. Practical LR error recovery. *SIGPLAN Notices 14*, 8, 168-175.

GRAHAM, S. L., AND RHODES, S. P. 1975. Practical syntactic error recovery. *Commun. ACM 18*, 11, 639-649.

GRAY, R. W. 1987. Automatic error recovery in a fast parser. USENIX Procs, Phoenix, 337-346.

GRIES, D. 1971. *Compiler Construction for Digital Computers.* Wiley, New York, NY.

GRIES, D. 1974. Error recovery and correction - an introduction to the literature. In *Compiler Construction: An Advanced Course*, Bauer and Eickel (Eds), Springer-Verlag, Berlin, 627-638.

GRUNE, D., AND JACOBS, C. J. H. 1988. A programmer-friendly LL(1) parser generator. *Softw. Pract. Exper. 18*, 1, 29-38.

HARPER, R., MILNER, R., AND TOFTE, M. 1988. The definition of Standard ML. ECS-LFCS-88-62, LFCS, Dept of Computer Science, University of Edinburgh, Edinburgh.

HAMMOND, K., AND RAYWARD-SMITH, V. J. 1984. A survey on syntactic error recovery and repair. *Comput. Lang. (Elmsford, NY) 9*, 1, 51-67.

HARTMANN, A.C. 1977. *A Concurrent Pascal Compiler for Minicomputers*. Springer-Verlag, Berlin.

HORNING, J.J. 1974. What the compiler should tell the user. In *Compiler Construction: An Advanced Course*, Bauer and Eickel (Eds), Springer-Verlag, Berlin, 525-548.

HORSPOOL, R.N, AND WHITNEY, M. 1990. Even faster LR parsing. *Softw. Pract. Exper. 20*, 6, 515-535.

IRONS, E.T. 1963. An error-correcting parse algorithm. *Commun. ACM 6*, 11, 669-673.

JAMES, L.R. 1972. A syntax directed error recovery method. Technical Report CSRG-13, Computer Systems Research Group, University of Toronto, Toronto.

JOHNSON, S. C. 1975. Yacc - yet another compiler-compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ.

JOHNSON, S. C. 1978. A portable compiler - theory and practice. *Conf. Record Fifth Annual ACM Symposium on Principles of Programming Languages*, ACM, New York, NY, 97-104.

KANTOROWITZ, E., AND LAOR, H. 1986. Automatic generation of useful syntax error messages. *Softw. Pract. Exper. 16*, 7, 627-640.

KERNIGHAN, B. W., AND RITCHIE, D. M. 1988. *The C Programming Language*, 2nd Edition. Prentice Hall, Englewood Cliffs, NJ.

KNUTH, D.E. 1968. Semantics of context-free languages. *Math. Syst. Theory 2*, 2, 127-145.

KOSKIMIES, K. 1990. Lazy recursive descent parsing for modular language implementation. *Softw. Pract. Exper. 20*, 8, 749-772.

KOSKIMIES, K., ELOMAA, T., LEHTONEN, T., AND PAAKKI, J. 1988a. TOOLS/HLP84 report and user manual. Report A-1988-2, Dept of Computer Science, University of Helsinki, Helsinki.

KOSKIMIES, K., NURMI, O., PAAKKI, J., AND SIPPU, S. 1988b. The design of a language processor generator. *Softw. Pract. Exper. 18*, 2, 107-135.

KRAWCZYK, T. 1980. Error correction by mutational grammars. *Inf. Process. Lett. 11*, 1, 9-15.

LEINIUS, R. P. 1970. Error detection and recovery for syntax directed systems. Ph. D. Thesis, Computer Science Department, University of Wisconsin, Madison.

LEVY, J.P. 1975. Automatic correction of syntax errors in programming languages. *Acta Inf. 4*, 3, 271-292.

LEWI, J., DE VLAMINCK, K., HUENS, J., AND HUYBRECHTS, M. 1978. The ELL(1) parser generator and the error recovery mechanism. *Acta Inf. 10*, 3, 209-228.

LYON, G. 1974. Syntax-directed least-errors analysis for context-free languages: a practical approach. *Commun. ACM 17*, 1, 3-14.

MCKEEMAN, B. J. 1990. *A Compiler Generator*. Prentice-Hall, Englewood Cliffs, NJ.

MCKENZIE, B. J. 1990. LR parsing of CFGs with restrictions. *Softw. Pract. Exper. 20*, 8, 823-832.

MATTERN, F. 1984. An annotated bibliography on error handling in compilers. Sonderforschungsbereich 124, "VLSI und Parallelitat", Teilprojekt D1, Bericht Nr 18/84, Fachbereich Informatik, Universitat Kaiserslautern, Kaiserslautern.

MAUNEY, J., AND FISCHER, C. N. 1981. An improvement to immediate error detection in strong LL(1) parsers. *Inf. Process. Lett. 12*, 5, 211-212.

MAUNEY, J., AND FISCHER, C. N. 1982. A forward move algorithm for LL and LR Parsers. *SIGPLAN Notices 17*, 6, 79-87.

MAUNEY, J., AND FISCHER, C. N. 1988. Determining the extent of lookahead in syntactic error repair. *ACM Trans. Program. Lang. Syst. 10*, 3, 456-469.

MICKUNAS, M.D., AND MODRY, J. A. 1978. Automatic error recovery for LR parsers. *Commun. ACM 21*, 6, 459-465.

MOLICH, R., AND NIELSEN, J. 1990. Improving a human-computer dialogue. *Commun. ACM 33*, 3, 338-348.

MOULTON, P.G., AND MULLER, M. E. 1967. DITRAN - a compiler emphasizing diagnostics. *Commun. ACM 10*, 1, 45-52.

NORMAN, D.A. 1983. Design rules based on analyses of human error. *Commun. ACM 26*, 4, 255-258.

PAI, A., AND KIEBURTZ, R. B. 1979. Global context recovery: a new strategy. *SIGPLAN Notices 14*, 8, 158-167.

PAI, A., AND KIEBURTZ, R. B. 1980. Global context recovery: a new strategy for syntactic error recovery by table-driven parsers. *ACM Trans. Program. Lang. Syst. 2*, 1, 18-41.

PEMBERTON, S. 1980. Comments on an error recovery scheme by Hartmann. *Softw. Pract. Exper. 10*, 231-240.

PENNELLO, T.M., AND DEREMER, F. 1978. A forward move algorithm for LR error recovery. *Conf. Record Fifth Annual ACM Symposium on Principles of Programming Languages*, ACM, New York, 241-254.

PETERSON, T. G. 1972. Syntax error detection, correction and recovery in compilers. Ph. D. thesis, Stevens Institute of Technology.

POONEN, G. 1977. Error recovery for LR(k) parsers. *Information Processing 77*, North Holland Publishing Co., 529-533.

RAIHA, K. J. 1980. Experiences with the compiler writing system HLP. *Procs. of a Workshop on Semantics-Directed Compiler Generation,* Lecture Notes in Computer Science, Springer-Verlag, Berlin, 350-362.

RAIHA, K. J., SAARINEN, M., SARJAKOSKI, M., SIPPU, S., SOISALON-SOININEN, E., AND TIENARI, M. 1983. Revised report on the compiler writing system HLP78. Report A-1983-1, Dept of Computer Science, University of Helsinki, Helsinki.

REISS, S.P. 1990. Interacting with the FIELD environment. *Softw. Pract. Exper. 20*, 51, S1/89-S1/115.

REPS, T. W., AND TEITELBAUM, T. 1989. *The Synthesizer Generator - A System for Constructing Language-based Editors*. Springer-Verlag, New York, NY.

RIPLEY, D.C., AND DRUSEIKIS, F. C. 1978. A statistical analysis of syntax errors. *Comput. Lang. (Elmsford, NY)* **3**, 227-240.

RÖHRICH, J. 1980. Methods for the automatic construction of error correcting parsers. *Acta Inf.* **13, 2**, 115-139.

SHNEIDERMAN, B. 1982. Designing computer system messages. *Commun. ACM 25*, 9, 610-611.

SIPPU, S. 1981. Syntax error handling in compilers. Report A-1981-1, Dept of Computer Science, University of Helsinki, Helsinki.

SIPPU, S., AND SOISALON-SOININEN, E. 1982. Practical error recovery in LR parsing. *Conf. Record Ninth Annual ACM Symposium on Principles of Programming Languages*, ACM, New York, NY, 177-184.

SIPPU, S., AND SOISALON-SOININEN, E. 1983. A syntax-error handling technique and its experimental analysis. *ACM Trans. Program. Lang. Syst. 5*, 4, 656-679.

STALLMAN, R.M. 1989. Using and porting GNU CC. Free Software Foundation Inc., Cambridge, MA.

SPENKE, M., MUHLENBEIN, H., MEVENKAMP, M., MATTERN, F., AND BEILKEN, C. 1984. A language independent error recovery method for LL(1) parsers. *Softw. Pract. Exper. 14*, 11, 1095-1107.

STIRLING, C. 1985. Follow set error recovery. *Softw. Pract. Exper. 15*, 3, 239-257.

TAI, K.C. 1978. Syntactic error correction in programming languages. *IEEE Trans. Softw. Eng. SE-4*, 5, 414-423.

TAI, K.C. 1980a. Locally minimum-distance correction of syntax errors in programming languages. *Procs. ACM National Conf.*, ACM, New York, NY, 204-210.

TAI, K.C. 1980b. Predictors of context-free grammars. *SIAM J. Comput. 9*, 3, 653-664.

TURNER, D. A. 1977. Error diagnosis and recovery in one pass compilers. *Inf. Process. Lett. 6*, 4, 113-115.

WAGNER, R. A., AND FISCHER, M. J. 1974. The string-to-string correction problem. *J. ACM 21*, 1, 168-173.

WETHERELL, C. 1977. Why automatic error correctors fail. *Comput. Lang. (Elmsford, NY) 2*, 179-186.

WIRTH, N. 1968. PL360 - a programming language for the 360 computers. *J. ACM 15*, 1, 37-74.

WIRTH, N. 1971. Design of a Pascal compiler. *Softw. Pract. Exper. 1*, 4, 309-333.

WIRTH, N. 1976. *Algorithms + Data Structures = Programs*. Prentice-Hall, Englewood Cliffs, NJ, 1976.